DEFENCE
SCIENCE & TECHNOLOGY

Sep 2002

Situation Description Language

S. Greenhill and S. Venkatesh and
A. Pearce and T.C.Ly

DSTO-GD-0332

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

DEFENCE
SCIENCE & TECHNOLOGY

Situation Description Language

S. Greenhill and S.Venkatesh

Curtin University of Technology

A. Pearce

University of Melbourne

T.C. Ly

Maritime Operation Division

Systems Sciences Laboratory

DSTO–GD–0332

ABSTRACT

SDL is a Situation Description Language intended for use in situation assessment problems. SDL provides knowledge modelling and inference facilities for reasoning with information.

The SDL includes an object-oriented data model with single inheritance. It provides a forward-chaining inference system using a RETE-based pattern matcher. SDL includes special data types to deal with time, space, and uncertainty. A hypothesis system allows the system to deal with multiple concurrent hypotheses in a systematic way.

This document provides details required by programmers or knowledge engineers intending to use the SDL system.

Published by

DSTO Systems Sciences Laboratory
PO Box 1500
Edinburgh, South Australia, Australia 5111

Telephone: *(08) 8259 5555*
Facsimile: *(08) 8259 6567*

AR No. 012-415
August, 2002

APPROVED FOR PUBLIC RELEASE

Situation Description Language

EXECUTIVE SUMMARY

Situation assessment is an essential process prior to making a decision. On submarines and other military platforms the operators take information from available sensors, and their background knowledge to deduce the tactical situation. Designing systems to replicate this process will give a better understanding of the process itself, and opens the possibility of automating the tasks that computers can perform better. The process is an information intensive process, requiring a high level language with concepts like those found in the field of artificial intelligence. The development of the Situation Description Language (SDL) brings together various techniques useful for representing the assessment process.

The SDL encompasses techniques for:

- Object-oriented data modelling with support for type-bound procedures and single-inheritance. This structures knowledge in a hierachical fashion. It also leads to simpler design, and allows common errors to be detected prior to execution.
- A forward-chaining inference system with RETE-based pattern matcher. This provides an efficient engine to do reasoning.
- A procedural programming system to allow encoding of procedures.
- Representations for time and space. This is vital for representing sensory information, which is temporal and spatial in nature.
- Representations for uncertainty, including a system for handling multiple concurrent hypotheses. The fog of war taints all information with some level of uncertainty.

This document provides details required by programmers or knowledge engineers intending to use the SDL system.

Contents

Appendices

1 Introduction

This document describes the semantics of SDL, a situation description language. SDL is a language intended to model knowledge within simulation environments, and provide a framework for reasoning with this information.

It includes:

- Object-oriented data modelling with support for type-bound procedures and single-inheritance.
- A forward-chaining inference system with RETE-based pattern matcher (7).
- A procedural programming system based loosely on the language Oberon-2.
- Representations for time (5.5) and space (5.6).
- Representations for uncertainty (`VARIANT` 5.4.2, `POTENTIAL` 5.2.3), including a system for handling multiple concurrent hypotheses (7.4).
- Parameterised types for sets and sequences (5.2).

This report uses an extended Backus-Naur Formalism (EBNF) to describe the syntax of `SDL`. Alternatives are separated by | Square brackets [and] indicate that the enclosed expression is optional. Braces { and } denote repetition (possibly 0 times) of the enclosed expression. Terminal symbols are enclosed within string quotes (eg. `"END"`).

The SDL had been implemented as a compiler and interpreter using Java. Description of that implementation is found in [1]. The SDL was used to model situation assessment in a submarine domain [2].

2 Tokens

```
ident = letter { digit | letter }.
number = digit { digit}.
realNumber =
  digit {digit} "." {digit} [ "E" [ "+" | "-"] digit {digit} ].
string =
  singleQuote { bodySingleString } singleQuote
| doubleQuote { bodyDoubleString } doubleQuote.
```

Identifiers are alphanumeric sequences that begin with a letter. A sequence of digits is interpreted as an integer, unless it is followed by a decimal point which indicates the start of a real number. Strings are sequences of characters delimited by either single or double quotes. The body of the string may contain any character except the delimiter. Examples:

```
3                (* integer *)
3.               (* real *)
3.1
```

```
6.022E24
"I'm a string"    (* string *)
'Another string'
```

3 Constant Declarations

A constant declaration associates an identifier with a constant value.

```
ConstantDeclaration = Ident "=" Expr.
ConstantExpr = Expr.
```

A constant expression can be evaluated at compilation time without actually executing part of a program. Examples:

```
CONST
  soundVelocity = 300;  (* m / second *)
```

4 Variable Declarations

Variable declarations associate an identifier with a data type.

```
IdentList = Ident { "," Ident }.
VariableDelaration = IdentList ":" Type.
```

A variable is a symbol that can change its value via assignment statements (8.1). Examples:

```
VAR
  i, j : INTEGER;
  s : SEQUENCE OF STRING;
  state : State;
```

5 Type Declarations

```
Type =
  Qualident
| SimpleType
| CollectionType
| CompoundType
| EnumeratedType
| "JAVA" JavaClassIdent
| "SPATIAL" | "POINT" | "LINE".

TypeDecl = Ident "=" Type.
```

Every data type defines a set of values which variables of that type may assume, and a set of operator that can be applied. A "type declaration" associates an identifier with a type. In the case of structured types such as collections (5.2) and compound types (5.4), it also defines the structure of variables of that type.

Examples:

```
(* instantaneous state of a contact *)
State = RECORD
  time : TIME;              (* time at which state was evaluated *)
  location : Position;      (* location of contact *)
  status : VARIANT          (* status of contact *)
    new;
    held;
    lost : LossReason;
  END;
END;
```

5.1 Simple Types

```
SimpleType =
  "BOOLEAN" | "CHAR" | "STRING" | "INTEGER" | "REAL".
```

The following simple types are defined:

BOOLEAN	the truth values TRUE and FALSE
CHAR	the set of ASCII characters
STRING	the set of finite length strings of CHARs.
INTEGER	the set of integers
REAL	the set of real numbers

5.1.1 Numeric Types

Arithmetic operators may be applied to operands of numeric type (ie. REAL or INTEGER). Integers may be used whenever a real number is required; they are automatically converted to real type. The following table outlines the operators defined for numeric types. Most symbols have an alternative "verbose" form. The DIV and MOD operators only apply to INTEGER values.

Symbol	Result Type	Meaning
+	Numeric	sum
-	Numeric	difference
*	Numeric	product
/	REAL	real quotient
DIV	INTEGER	integer quotient
MOD	INTEGER	integer modulus
<	BOOLEAN	less than
<=	BOOLEAN	less than or equal to
=	BOOLEAN	equal to
#	BOOLEAN	not equal to
>=	BOOLEAN	greater than or equal to
>	BOOLEAN	greater than

Numeric and relational operators have "verbose" equivalents, which can be used to write expressions that are closer to natural language.

Operator	Alternative
+	PLUS
-	MINUS
*	TIMES
	MULTIPLIED BY
/	DIVIDED BY
<	LESS THAN
=	EQUAL TO
	EQUALS
#	NOT EQUAL TO
>	GREATER THAN

In this example, the following are equivalent expressions:

```
5 * n + 1 < y
1 PLUS 5 MULTIPLIED BY n LESS THAN y
```

5.1.2 String Operators

The following operators apply to strings.

Symbol	Result Type	Meaning
a + b	STRING	concatenation of arguments
a[i]	CHAR	character at position i in string a
SIZE(a)	INTEGER	number of characters in string a
<	BOOLEAN	less than
<=	BOOLEAN	less than or equal to
=	BOOLEAN	equal to
#	BOOLEAN	not equal to
>=	BOOLEAN	greater than or equal to
>	BOOLEAN	greater than

A CHAR may be used wherever a string is required; it is automatically converted to a string of length 1.

5.1.3 Character Operators

The following operators apply to characters.

Symbol	Result Type	Meaning
<	BOOLEAN	less than
<=	BOOLEAN	less than or equal to
=	BOOLEAN	equal to
#	BOOLEAN	not equal to
>=	BOOLEAN	greater than or equal to
>	BOOLEAN	greater than

Characters are returned as elements of strings using the index operator. Characters may also be constructed using the CHR operator, which returns the character at a given ordinal position in the ASCII table. For example:

```
PRINTLN CHR(65) = "A";
```

5.1.4 Boolean Operators

The following operators apply to Boolean values. All Boolean operators return Boolean results.

Symbol	Meaning
p OR q	if p then TRUE else q
p AND q	if p THEN q ELSE FALSE
NOT p	not p
=	equal to
#	not equal to

In expressions involving Boolean operators, sub-expressions are only evaluated if their value can influence the result of the expression. This is known as "conditional" or "Short-Circuit" evaluation.

Expressions of the form:

```
C1 AND C2 AND ... AND Cn     C1 OR C2 OR ... OR Cn
```

are treated as

```
IF ~C1 THEN                  IF C1 THEN
  FALSE                        TRUE
ELSIF ~C2 THEN               ELSIF C2 THEN
  FALSE                        TRUE
ELSIF ...                    ELSIF ...
```

```
ELSE                    ELSE
  Cn                      Cn
END                     END
```

Conditional evaluation is used for all Boolean expressions. It is also used for rule conditions.

Example:

```
C1 AND (C2 OR C3) OR C4
```

In the example, the possible orders of evaluations are:

```
C1 (false), C4
C1 (true) C2 (true)
C1 (true) C2 (false) C3 (true)
C1 (true) C2 (false) C3 (false) C4
```

5.2 Collection Types

```
CollectionType =
  "SET" "OF" Type
| "SEQUENCE" "OF" Type
| "POTENTIAL" Type.
```

A collection contains a number of elements of the same type. The number of elements in a collection is called its "size". A SET is a collection where each element is unique. A SEQUENCE is a collection where elements may be duplicated, and the order of the elements is significant. A SEQUENCE type may optionally specify a size. A POTENTIAL type is a collection where each element is unique and is associated with a REAL "potential", which may be interpreted as the confidence or certainty of set membership.

Example:

```
TYPE GivenNames = SEQUENCE OF STRING;
VAR daysPerMonth : SEQUENCE 12 OF INTEGER;
TYPE History = SEQUENCE OF State;
```

The elements of a collection are designated by indices, which are integers between 0 and the size minus 1. Thus, to iterate over the elements of a sequence s, the following can be used:

```
FOR i := 0 TO SIZE(s)-1 DO
  ... something with s[i] ...
END
```

The FOREACH statement iterates over elements of any collection (either SEQUENCE or SET). For example,

```
FOREACH e IN c DO
  ... something with e ...
END;
```

5.2.1 Sets

```
Element = Expr.
ElementList = [ Element { "," Element } ].
Set = "{" ElementList "}".
```

SET values are constructed by enclosing their elements within braces. For example:

```
{ "a", "b", "c", "d" }        (* a SET OF STRING *)
{ 1, 3, 5, 7, 9 }             (* a SET OF INTEGER *)
{ { 2, 4, 6 }, { 1, 3, 5 } }  (* a SET OF SET OF INTEGER *)
{}                            (* an empty set *)
```

The following operators apply to SET types.

Operator	Result Type	Meaning	
a + b	SET	set union	$a + b = \{x \mid x \in a \vee x \in b\}$
a * b	SET	set intersection	$a * b = \{x \mid x \in a \wedge x \in b\}$
a - b	SET	set difference	$a - b = \{x \mid x \in a \wedge x \notin b\}$
a / b	SET	set symmetric difference	a/b = (a-b) + (b-a)
e IN a	BOOLEAN	membership	true when element e is in set a

Example:

```
a := {1, 2, 3, 4}; b := {1, 3, 5, 7};

a + b = {1, 2, 3, 4, 5, 7}
a - b = {2, 4}
a * b = {1, 3}
a / b = {2, 4, 5, 7}
2 IN a = TRUE
SIZE(a) = 4
```

5.2.2 Sequences

```
Sequence = "[" ElementList "]".
```

SEQUENCE values are constructed by enclosing their elements within square brackets. For example:

```
[ "a", "b", "c", "d" ]        (* a SEQUENCE OF STRING *)
[ 1, 3, 5, 7, 9 ]             (* a SEQUENCE OF INTEGER *)
[ [ 2, 4, 6 ], [ 1, 3, 5 ] ]  (* a SEQUENCE OF SEQUENCE OF INTEGER *)
[]                            (* an empty sequence *)
```

The following operators apply to SEQUENCE types.

Operator	Result Type	Meaning	
a + b	SEQUENCE	concatenation	
e IN a	BOOLEAN	membership	true when element e is in sequence a
SIZE(a)	INTEGER	size	length of the sequence a

Example:

```
a := [1, 2, 3, 4]; b := [1, 3, 5, 7];

a + b = [1, 2, 3, 4, 1, 3, 5, 7]
a[2] = 3
SIZE(a) = 4
```

5.2.3 Potential types

```
PotentialElement = Expr "CF" Expr.
PotentialElementList = [ PotentialElement { "," PotentialElement } ].
PotentialSet = "{{" PotentialElementList "}}".
```

POTENTIAL values are constructed by enclosing their elements within double braces. For each element, a certainty factor must be given. For example:

```
{{ "a" CF 0.4, "b" CF -0.2 }}       (* a POTENTIAL STRING *)
{{ 1 CF 0.1, 3 CF 0.1, 5 CF 0.1 }}  (* a POTENTIAL INTEGER *)
{{}}                                (* an empty POTENTIAL set *)
```

The following operators apply to POTENTIAL types.

Operator	Result Type	Meaning	
a + b	POTENTIAL	union	
e IN a	BOOLEAN	membership	potential of element e in set a
SIZE(a)	INTEGER	size	number of values in a
LIKELY	SET	most likely	return the set of values with maximal potential.
UNLIKELY	SET	least likely	return the set of values with minimal potential.

Union of potential values is similar to set union. Where a value occurs in both operands, its potential is defined by the following rule[1]:

$$Combine(a,b) = \begin{cases} a + b(1-a) & a \geq 0, b \geq 0 \\ a + b(1+a) & a < 0, b < 0 \\ \frac{a+b}{1-\min(abs(a),abs(b))} & \text{otherwise} \end{cases}$$

According to this definition, an element can always be removed from a set by union with a set in which its potential is negated. For example:

[1]This rule originates from the representation of certainty used in the MYCIN expert system

```
a := a + {{ e CF - (e IN a) }};     (* remove e from a *)
```

Examples:

```
a := {{"John" CF 0.2, "Jane" CF 0.4}};

SIZE(a) = 2
"John" IN a = 0.2
"Jim" IN a = 0.0
```

Note that white-space is significant in distinguishing between potential values and nested sets.

```
SIZE({{}}) = 0          (* empty POTENTAL value *)
SIZE({ {} }) = 1        (* SET containing an empty SET *)
```

An empty collection has undefined type, but is compatible with a corresponding collection of any base type.

5.3 Enumerated Types

```
EnumeratedType = "(" IdentList ")".
```

An enumerated type is defined by a finite set of identifiers that represent legal values of that type. This construct is most often used to express a set of options for choice. For example:

```
TYPE
  Day = ( Mon, Tue, Wed, Thu, Fri, Sat, Sun );
  Vowel = ( A, E, I, O, U );
  Note = ( A, B, C, D, E, F, G );
```

An identifier may be a member of more than one enumeration. In this case its type is undefined, but is compatible with any enumeration of which it is a member. If T is an enumerated type, the identifier T has type SET OF T and denotes the set of all members of T. Therefore, the following are legal:

```
PRINTLN Vowel;
PRINTLN A IN Vowel;
PRINTLN {A, E} * Vowel;
PRINTLN {A, E} * Note;
```

but the following are errors:

```
PRINTLN B IN Vowel;     (* incompatible types *)
PRINTLN Vowel * Note;   (* incompatible types *)
```

There is no defined ordering on the elements of an enumerated types (eg. as in Pascal). However, it is possible to enumerate the values using:

```
VAR e : T;
...
FOREACH e IN T DO ... END;
```

See also: VARIANT types (5.4.2).

5.4 Compound Types

```
Fields = [ IdentList [ ":" Type] [ "DEFAULT" Expr ] ] .
FieldList = Fields { ";" Fields }.
CompoundType =
 { "ABSTRACT" | "HYPOTHETICAL"
 | "INSTANT" | "INTERVAL" | "PERSISTENT" }

 ( "RECORD" | "VARIANT")

 [ "(" Qualident ")" ]

 FieldList
"END" .
```

"Compound types" specify new types in terms of a fixed number of elements called "attributes". Any compound type (RECORD 5.4.1, VARIANT 5.4.2) may be declared as an extension of an existing type.

Example:

```
Point2D = RECORD
  x, y : REAL
END;

Point3D = RECORD (Point2D)
  z : REAL
END;
```

In the declaration, Point3D is an "extension" of Point2D; Point2D is a "base type" of Point3D. An extended type includes all attributes of its base type. An extended type may have only one base type.

If type A is "compatible" with type B, a value of type A may be used whenever type B is required. In general, A is considered to be "type compatible" with B only when A is guaranteed to contain all the attributes of B. Thus, an extension of a RECORD type is "type compatible" with its base types. A base type of a VARIANT is "type compatible" with its extensions.

Type compatibility means that the dynamic type of a variable may be different from its static type. Three forms of type test can be used to test the dynamic type of a value.

A "type guard" is a designator of the form:

```
v{T}
```

The type guard asserts that v has dynamic type that is "type compatible" with T, and treats v as having static type T within the designator.

A "type test" is an expression of the form:

```
v IS T
```

The value of the expression is true only when the dynamic type of v is "type compatible" with T.

An "attribute test" is an expression of the form:

```
v HAS a
```

The value of the expression is true only when v has attribute a. The HAS operator is typically used for VARIANT values. It is trivially TRUE for all RECORD values.

The following operators apply to all compound types.

Operator	Result Type	Meaning	
v.a	field type	field selection	attribute a of value v
v IS T	BOOLEAN	type test	true when dynamic type of v is type compatible with type T
v HAS a	BOOLEAN	attribute test	true when v has attribute a
v(T)	T	type guard	asserts that the dynamic type of v is type compatible with type T, and treats the designator as having static type T

The value NIL may be given whenever a compound value is expected. This signifies an undefined or invalid object. Attempting to apply the above operators to a NIL value results in a run-time error.

A compound type may be declared as ABSTRACT, which means that it may not be instantiated. Abstract types are used only as base types for other types.

The name of any compound type may be used as a designator for the set of its instances. This allows queries to be expressed using set operators. See SELECT expressions (6.4).

5.4.1 RECORD Types

RECORD types are used to represent composition. An object may be defined to be the combination of a number of component attributes.

A RECORD type is a compound type consisting of a fixed number of named "attributes" each of a declared type. In a RECORD value, all of the named attributes exists independently and may be selected by name (see Designators).

Example:

```
Person = RECORD
  surname : STRING;
  givenNames : SEQUENCE OF STRING;
  age : INTEGER
END;

Employee = RECORD (Person)
  salary : REAL
END;
```

A value p of type Person has three attributes: surname, givenNames, and age. A value e of type Employee has four attributes: salary is added to the three attributes of its base class. Type guards may be used when an operation depends on the dynamic type of an object. For example:

```
p.age := 21;
IF p IS Employee THEN
  p(Employee).salary := 10000;
END;
```

5.4.2 VARIANT Types

VARIANT types are used to represent choice.

A VARIANT type is a compound type consisting of a fixed number of named "attributes" each of a declared type. In a VARIANT value, only one of the named attributes is valid at any time. Every VARIANT value has a "dynamic" attribute which is the only valid attribute among the alternatives.

Example:

```
SonarShift = RECORD
  amount : ( low, high );
  CPAposition : ( approaching, at, receding );
END;

SonarDoppler = VARIANT
  unShifted;
  shifted : SonarShift
END;
```

A value of type SonarDoppler has either attribute unShifted or attribute shifted, but not both. Like RECORDs, attributes may be selected by name. If an attribute does not match the "dynamic" attribute of the VARIANT value, an exception is generated. To safely access VARIANT types, conditionals (see IF, WITH) should be used. The HAS operator (see HAS) can be used to test the "dynamic" attribute of a VARIANT value. For example, if s is of type SonarDoppler:

```
IF s HAS shifted THEN
```

```
  IF s.shifted.amount = high AND
     s.shifted.CPAPosition = approaching THEN
      (* the object is rapidly approaching CPA *)
  END
ELSE
  (* we don't have doppler information *)
END;
```

NEW (6.3) may be used to instantiate variant values, specifying both the attribute and value.

```
s := NEW SonarDoppler(
  :shifted NEW SonarShift(:amount high, :CPAPosition approaching));
```

For VARIANT type declarations it is legal to omit the element field type. Such elements are said to have *unity* type; their values are not significant. VARIANT may be extended in the same way as RECORDs.

The following example defines two variant types. SonarSensor enumerates all valid sonar sensor types. Sensor extends SonarSensor to include other types of sensor.

```
SonarSensor = VARIANT
  TA;                          (* Towed Array *)
  CA;                          (* Cylindrical Array *)
  FA;                          (* Flank Array *)
  DA;                          (* Distributed Array *)
  IA;
END;

Sensor = VARIANT (SonarSensor)
  ESM;                         (* ESM transmission *)
  visual;                      (* Periscope sighting *)
END;
```

See also: Enumerated types (5.3).

5.5 Temporal Data Types

If precise information is available, time may be specified as the location on the quantitative time axis in units of years, days, months, hours, minutes, seconds, and milliseconds. Such times can be encoded using REAL or INTEGER types and tests can be made using ordinary relational operators.

Alternatively, time may be represented in terms of constraints between abstract points in time. An *instant* represents an instantaneous point in time. Instants are not necessariliy related to a particular location on a temporal axis, and may be related qualitatively (eg. using relations like before, after, during). An *interval* represents a period of time. Every interval is bounded by two instants, its *start* and its *end*.

RECORD types may be tagged as INSTANT or INTERVAL to allow them to participate in temporal constraints.

5.5.1 Temporal Constraints

A temporal constraint expresses relationships between temporal objects (of type INTERVAL or INSTANT). Temporal constraints are expressed using a natural-language-like syntax.

```
TemporalDesignator = UnaryExpr.
TemporalAnchor = ( "STARTS" | "ENDS" | "HAPPENS").
TemporalUnit
= ( ("MILLISECOND" | "MS")
  | ("SECOND" | "SECONDS" )
  | ("MINUTE" | "MINUTES" )
  | ("HOUR" | "HOURS" )
  | ("DAY" | "DAYS" )
  ).
Time = "@" Expr {":" Expr } TemporalUnit.

AbsoluteTime =
  ( TemporalDesignator [TemporalAnchor ]
  | "START" "OF" TemporalDesignator
  | "END" "OF" TemporalDesignator).
```

A "temporal designator" is a unary expression that has a temporal data type. This may be a metric "time", or an attribute of an INSTANT or INTERVAL record designated by a "temporal anchor".

Times are denoted by the @ symbol. The "temporal unit" denotes the size of the most significant division of the time. Each : introduces one level of subdivision. For example:

```
  @ 08:30 HOURS                (* the time 8:30 AM *)
  @ 1999:07:25:08:30:11 YEARS  (* 8:30:11, 25 July 1999 *)
```

An "absolute time" describes a precise point in time (note that temporal designators may describe intervals). For example:

```
  @ 10:00 HOURS                (* the time 10:00 AM *)
  START OF e                   (* the time that interval e starts *)
  i HAPPENS                    (* the time that instant i happens *)

TemporalRange
= ( ["EXACTLY"] AddExpr TemporalUnit
  | "RANGE" Expr [ TemporalUnit ] "TO" Expr TemporalUnit
  | "MORE" "THAN" AddExpr TemporalUnit
  | "LESS" "THAN" AddExpr TemporalUnit
  ).

TemporalDistance
= ( TemporalRange | "SOON" | "SHORTLY" | "LONG" ).
```

```
TemporalConstraint
= ( TemporalAnchor
    (
      ("AT" | "AS") AbsoluteTime
    | [TemporalDistance]
      ( "AFTER"
      | "BEFORE"
      )
      AbsoluteTime
    | "DURING" TemporalDesignator
    | "BETWEEN" AbsoluteTime "AND" AbsoluteTime
    | "WITHIN" Expr TemporalUnit "OF" AbsoluteTime
    )
  | "HAS" "DURATION" TemporalRange
  ).
```

A "temporal constraint" describes the relationship between two events. Constraints define the duration of an interval, or relationships between an anchor of some event (ie. its start, end) and other points in time. A "temporal range" describes distance in time.

```
TemporalConstraints = TemporalConstraint { "ALSO" TemporalConstraint } .
TemporalStat =
  "TEMPORAL" [ "IN" "(" Expr ")" ] {
    TemporalDesignator TemporalConstraints ";"
  } "END".
```

A "temporal statement" is a collection of constraints that applies to one or more temporal objects. Optionally, an expression may be given to define the hypothesis within which the statement applies (see Hypotheses (7.4)). A temporal statement may appear as part of a temporal query or temporal assertion.

Examples:

```
TEMPORAL (* Temporal assertion *)
  johnWalk (* John walks to the bus stop *)
         STARTS BETWEEN @7:00 HOURS AND @7:10 HOURS
    ALSO HAS DURATION RANGE 5 TO 10 MINUTES;
  johnBus  (* John takes the bus to work *)
         STARTS RANGE 5 TO 10 MINUTES AFTER johnWalk ENDS
    ALSO HAS DURATION RANGE 20 TO 30 MINUTES;
  jimRide  (* Jim rides his bike to work *)
         STARTS BETWEEN @6:30 HOURS AND @6:45 HOURS
    ALSO HAS DURATION RANGE 40 TO 50 MINUTES;
  ping   HAPPENS AFTER bang HAPPENS
    ALSO HAPPENS DURING jimRide
    ALSO HAPPENS DURING johnWalk;
END;
```

```
PRINTLN TEMPORAL (* Query if John arrives after Jim? *)
  johnBus ENDS AFTER jimRide ENDS;
END;

PRINTLN TEMPORAL (* Query *)
  johnBus STARTS WITHIN 25 MINUTES OF END OF jimRide;
END;
```

5.6 Spatial Data Types

Spatial data types include the following:

```
SPATIAL
  POINT
  EXTENT
    LINE
```

The type SPATIAL is a base type for all two-dimensional spatial types. The type POINT type represents points. EXTENT is an abstract type containing spatial types that have "interiors" which may contain points. A LINE represents the path between a sequence of connected points.

5.6.1 Spatial Operators

The following spatial operators are defined:

Operator	Result Type	Meaning
DISTANCE FROM p TO q	REAL	Distance from point p to point q
BEARING FROM p TO q	REAL	Direction from point p to point q
LENGTH OF l	REAL	Length of line l
CLOSEST TO p OF s	POINT	The point in s that is closest to point p

5.7 Java Types

Java classes can be used within SDL with the following restrictions:

- Only Java class methods are available. It is not possible to use static procedures, or variables.
- Exceptions are not supported. If a Java class method throws an exception, the program will be terminated.
- A restricted set of types are available. Java types that correspond directly to SDL types may be used. Currently, SDL does not support any methods that would require translation of values at run-time.

  The table below shows the correspondence between SDL types and the allowed types in Java methods. Note that arrays are not supported.

Java Type	SDL Type
int, or java.lang.Integer	INTEGER
java.lang.String	STRING
boolean, or java.lang.Boolean	BOOLEAN
double, or java.lang.Double	REAL
Object type T	JAVA T

There are a few differences between the type systems of SDL and Java that can affect the usefulness of Java interfaces. Some SDL constructs have no corresponding equivalent in Java, so it will not always be possible to obtain the desired method signature in a pure Java declaration. For example, SDL has a parametric SET type whereas Java sets have a single member class `java.lang.Object`.

One way around this problem is to use SDL `HINT` declarations. These inform the SDL compiler of the correct interpretation for methods. For example:

```
HINT SDL.SpatialLine (Intersect) : SET OF SPATIAL;
```

This informs the SDL compiler that the return type of the `Intersect` method of the `SDL.SpatialLine` class is `SET OF SPATIAL`. Without this declaration, the result type determined from the method signature would be `java.util.Set`.

Currently the `HINT` mechanism is restricted to return type declarations, although it could usefully be extended to handle the whole method signature including method parameters.

The following example illustrates the use of Java classes within SDL code.

```
PROCEDURE FetchURLString (name : STRING) : STRING;
(* Attempt to fetch data from URL <name>. The resulting text is returned
 * as a STRING *)
TYPE
  StringBuffer = JAVA java.lang.StringBuffer;
  InputStream = JAVA java.io.InputStream;
  URL = JAVA java.net.URL;
VAR
  i : InputStream;     (* input from server *)
  c : INTEGER;         (* character read from input *)
  s : StringBuffer;    (* result string *)
BEGIN
  (* open stream of input from URL *)
  i := NEW URL(name) .openConnection() .getInputStream();
  s := NEW StringBuffer();

  (* fetch data from the stream, and append to string *)
  c := i.read();
  WHILE c # -1 DO
    s := s.append(CHR(c));
    c := i.read()
  END;
```

```
  i.close();
  RETURN s.toString();
END FetchURLString;
```

6 Expressions

Expressions are ways of combining values by the application of operators and functions.

6.1 Operands

Operands are values to which operators are applied. The simplest operand is a literal constant, such as a string or number. Structured values such as collections (5.2) and compound types (5.4) can be constructed dynamically; these constructors are also operands. Otherwise, operands are denoted by "Designators". A designator consists of an identifier referring to a constant, variable, or procedure. If the designator refers to a structured object, it may be followed by "selectors" which identify an element of the structure. Within temporal expressions (5.5.1), "pattern selectors" may be used to designate objects that match patterns. This form of designator is only used with rules (7).

```
FieldSelector =  "." Ident .
ArraySelector =  "[" Expr "]" .
FunctionSelector = "(" [ ActualParam { "," ActualParam } ] ")" .
TypeSelector = "{" Type "}" .

Selector =
    FieldSelector
  | ArraySelector
  | FunctionSelector
  | TypeSelector .

Designator =
( Qualident
| NewExpression
| "MAP" "(" Expr "," Expr ")"
)
{ Selector } .

Selector =
    "." Ident
  | "[" Expr "]"
  | "(" [ ActualParamList ] ")".
```

If v designates a compound value, then v.a designates attribute a of v, or the procedure a bound to the dynamic type of v. For variant values, an error occurs if a is not the dynamic attribute of v.

If v designates a sequence value, then v[e] designates the element of v whose index is the current value of the expression e. If there is no such element (the index is greater than the length of the sequence) an error is signalled.

A type guard v{T} asserts that v has dynamic that is type compatible with T, and treats v as having static type T within the designator. If the dynamic type of v is not type compatible with T, an error is signalled. If T is not type compatible with the static type of v, the type guard always fails: this situation is signalled as an error during compilation.

If the designated object is a variable, the designator refers to its current value. If it is a procedure and is followed by a (possibly empty) parameter list, it implies an activation of the procedure and represents the value resulting from its execution. The actual parameters must correspond the formal parameters (9.1).

Examples:

```
i
Module.i
name[i]
t.next.prev
p{Employee}.salary
p.Show(this)
```

6.2 IF Expressions

```
IfExpression =
  "IF" Expr "THEN" Expr
  {"ELSIF" Expr "THEN" Expr }
  "ELSE" Expr "END"
```

IF expressions specify conditional evaluation of expressions. A Boolean expression called a "guard" precedes each expression. The guards are evaluated in sequence until one evaluates TRUE, whereupon its associated expression is returned. If no guard is satisfied, the expression following the ELSE symbol is returned.

All non-guard expressions in an IF expression must be of the same type. The expression is said to have this same type.

See also: IF Statements (8.4).

6.3 NEW Expressions

```
InitialParam = [":" Ident] [ Expr ] .
Initialisation =
  ( "JAVA" JavaClassIdent | Qualident )
  "(" InitialParam { "," InitialParam } ")".
NewExpression = "NEW" Initialisation.
```

NEW expressions return new instances of a type. The type may be a named SDL type, or a Java type prefixed by JAVA.

The parameters to the new expression are treated differently depending on what kind of type is being instantiated.

For SDL types the parameters define the values of fields of the compound type. The names of the fields must be specified, but the order is not significant. Fields not specified are undefined in the newly created record. Some fields must be specified (eg. see hypotheses 7.4) or a compile-time error results.

Example:

```
TYPE Complex = RECORD x, y : REAL END;
...
VAR c : Complex;
...
c := NEW Complex(:x 0, :y 0);
```

For Java types, the parameters define parameters to a constructor procedure. The names of the parameters are not specified, but order is significant.

Example:

```
PRINTLN NEW java.lang.String("hello");

s := NEW JAVA java.net.URL(name).openConnection().getInputStream();
```

See also: CREATE (8.9), DELETE (8.12).

6.4 SELECT Expressions

```
SelectWhere = [ "LET" Ident Becomes ] Expr .
SelectFrom = Ident ":" Expr .

FromWhere = SelectFrom { "," SelectFrom }
  [ "WHERE" SelectWhere { "," SelectWhere } ] .

SelectExpr = "FROM" FromWhere "SELECT" Expr "END" .
```

SELECT expressions are used to filter or combine sets of objects. The result of a SELECT expression is a sequence of values.

The "from" part specifies a number of expressions, each of which must denote a set. A variable identifier is associated with each set. The identifier is used to refer to an object within the set.

The "where" part gives a set of expressions which must be TRUE for values to be returned. These may be interspersed with "let" declarations, which assign temporary labels for values.

The "select" part gives an expression that is evaluated in the context of the "where" part to generate each member of the output set. The result of the "select" expression is a set

with the base type of the "select" expression. It is therefore possible to return structured values by giving an expression that invokes a NEW operator or sequence constructor.

The simplest application of the SELECT statement is for filtering sets based on selection criteria. For example, the following selects objects of type LandForm where the closest point of the object's location is less that 1000m from pos:

```
SELECT l FROM l : LandForm
WHERE
  pos.DistanceTo(pos.ClosestTo(l.location)) < 1000
END;
```

It is possible to form more complex queries that relate objects in multiple sets, and return structured values. For example, the following returns a list of pairs of integers from two source sets where the first is less than the second. In the process, it also calculates their sum and difference.

```
FROM
  x : [1, 3, 5], y : [2, 4, 6]
WHERE
  x < y,
  LET sum := x + y,
  LET diff := x - y
SELECT
  [x, y, sum, diff]
END;
```

The operation yields the result:

```
{ [5, 6, 11, -1],
  [1, 2, 3, -1],
  [3, 6, 9, -3],
  [3, 4, 7, -1],
  [1, 6, 7, -5],
  [1, 4, 5, -3]}
```

The following example computes a relationship between multiple members of the same set.

```
TYPE
  Sex = (Male, Female);

  Person = RECORD
    name : STRING;
    sex : Sex;
    parent : SET OF STRING;     (* names of parents *)
  END;
```

```
...

PROCEDURE GrandParents (sex : Sex) : SET OF Person;
BEGIN
  RETURN
    FROM
      a : Person, b : Person, c : Person
    WHERE
      a.sex = sex,
      a.name IN b.parent,
      b.name IN c.parent
    SELECT
      a
    END;
END GrandParents;
```

See also: EXISTS (6.5) and ALL (6.5).

6.5 EXISTS / ALL Expressions

```
ExistExpr = ( "EXISTS" | "ALL" ) FromWhere "END" .
```

The EXISTS and ALL expressions are closely related to the SELECT expression.

An EXISTS expression is satisfied if some expression in the "where" part evaluates true. An ALL expression is satisfied if all expressions in the "where" part evaluate true. Therefore, except for performance:

```
EXISTS t : T WHERE f(t) END;
```

is equivalent to

```
FROM t : T WHERE f(t) SELECT t END # {};
```

Similarly,

```
ALL t : T WHERE f(t) END;
```

is equivalent to

```
FROM t : T WHERE ~f(t) SELECT t END = {};
```

The EXISTS and ALL forms are inherently more efficient because they do not generate result sets, and terminate the iteration of sources as soon as satisfiability is determined.

See also: SELECT (6.4).

7 Rules

Rules are used to trigger actions in response to patterns of data.

```
ForwardRule = "IF" Cond "THEN" RuleStatementList .

EventRule =
  "EVENT" Pattern "WHEN" Cond
  [ "ACTIVE" TemporalConstraints ]
  [ "INACTIVE" TemporalConstraints ] .

RuleDecl = "RULE" Ident (ForwardRule | EventRule) "END" Ident .
```

"Forward chaining" rules consist of a "condition" and an "action." For the rule to fire, the condition must be satisfied by matching against existing data. The "action" is a set of statements which are executed within the context of the condition.

"Event" rules control the generation of EVENT objects. When the "from condition" of the event rule is satisfied, an event of the given "pattern" is created using the context defined by the condition. The event remains "matchable" by other rules until its "until condition" is satisfied. When this occurs, the result depends upon whether the event type is declared PERSISTENT. If persistent, the event object is maintained in the data base but marked "inactive" which means that it no longer matches patterns in rules. If not declared persistent, the event object is deleted from the data base.

For non-persistent events, the event rule:

```
RULE Name
EVENT
  Pattern
WHEN
  Condition
END Name;
```

is equivalent to a pair of rules:

```
RULE AssertName
IF
  Condition & ~ Pattern
THEN
  CREATE Pattern
END AssertName;

RULE RetractName
IF
  Pattern p & ~ Condition
THEN
  DELETE p;
END RetractName;
```

The following event rule creates a `ProximityOfTOI` event when an `Entity` is found to be in the vicinity of the expected location of a target of interest (`TOI`). The rule fires whenever the location of the entity changes and the event is not already in place.

```
RULE R3
  EVENT
    ProximityOfTOI {entity <e> toi <t>}
  WHEN
    Entity e { location <l> } &
    TOI t { : l INSIDE t.location }
END R3;
```

7.1 Patterns

```
PatternBinding = Ident ( "<<" Ident ">>" | "<" Ident ">" | Expr).

Pattern = Qualident [ Ident ] "{" { PatternBinding | ":" Expr } "}" .
```

A Pattern is composed of a type identifier, a possibly empty set of bindings, and an optional variable identifier. The pattern is satisfied when an object of the indicated type matches the bindings. If a variable identifier is given, this is bound to the object that matches the pattern.

Each binding specifies an attribute and the required value for that attribute. If the value is an expression, the attribute must match this value. If the value is a pattern variable, the attribute must match the value of the variable. A set of conditions is only satisfied if a set of bindings exists that satisfies each individual pattern. If one or more optional Boolean expressions are given, the match only succeeds if the expressions all evaluate true.

Within a pattern binding, identifiers enclosed within single brackets (eg. `<X>`) denote pattern variables which are bound to the value of the given attribute. Identifiers enclosed within double brackets (eg. `<<Y>>`) denote pattern variables which are bound to the value of an element of an attribute with collection type (ie. set, sequence, or potential).

Example:

```
Employee e1 {surname "Smith" salary <S> : S > 10000}
Person {givenNames <<N>> : N = "John"}
Person {givenNames <M> : "John" IN M }
```

The first condition matches any Employee with a `surname` of `"Smith"` who has a `salary` greater than 10000. The variable `e1` is bound to the matching Employee. The second condition matches any `Person` having `"John"` as a given name. The third condition is equivalent.

7.2 Conditions

```
Term = "(" Cond ")" | Pattern .
```

```
NotCond = [ "~" ] Term .

AndCond = NotCond { "&" NotCond }.

OrCond = AndCond { "|" AndCond }.

Cond = OrCond.
```

"Conditions" are a special form of expressions that can be used to define when rules are to be applied. The fundamental form of condition is the "Pattern" (7.1) which matches objects in the data base. Conditions may be combined using logical operators similar to Boolean operators.

The "conditional and" `&` is satisfied whenever all of its sub-conditions are satisfied. The "conditional or" `|` is satisfied whenever one of its sub-conditions is satisfied. The "conditional negation" `~` is satisfied whenever its sub-condition is not satisfied. Logical conditions are implemented as operators. They obey the same precedence rules as Boolean operators: `~` has the highest precedence, followed by `&` and lastly `|`. Parentheses can be used to When conditions are separated by semicolons in a condition list, there is an implied "conditional and" operation, but this has a lower precedence than `|`.

Example:

```
C1 & (C2 | C3) | C4
```

Note that conditions are evaluated left to right with "conditional evaluation" semantics (5.1.4). If a condition does not affect the outcome, it is not evaluated.

The `FORALL` and `EXISTS` conditions are used for existential quantification. The statement

```
FORALL x : y
```

is equivalent to

```
~ (x & ~y)
```

Likewise,

```
EXISTS x : y
```

is equivalent to

```
~ ~ (x & y)
```

Note that the `|` an `~` operators have special considerations with respect to variable bindings. For the `|` operator, different variable bindings may result depending on which sub-condition is satisfied. For example:

```
Complex{real <r> : r > 1} | Complex{imag <i> : i > 1}
```

is satisfied by any Complex object that has a real attribute greater than 1 or a imag attribute greater than 1. Depending on the data, either r or i or both would be bound as a result. This situation is forbidden by the requirement that every sub-condition of a "conditional or" bind the same set of variables. This allows strict checking of variables at compile time.

For the ~ operator, the bindings in the sub-expression are never valid since it requires that the sub-expression not be satisfied. Thus, these bindings cannot be used in further conditions. This same restriction also applies to FORALL and EXISTS conditions, since these implicitly involve negation.

7.3 Example Rules

This section demonstrates the use of rules to find paths through a graph. It is intended to illustrate how to represent facts and goals using objects, and how to structure rules using patterns.

```
Room = RECORD
  name : STRING;
  id : INTEGER;
  exit : SET OF INTEGER;
END;
```

A map is represented as a set of Room objects. Each Room has an identifier id and a number of exits, exit, represented as a set of room identifiers. A map might look like this:

```
CREATE
  Room(:name "Entry",   :id 1, :exit {2});
  Room(:name "Hallway", :id 2, :exit {4, 3});
  Room(:name "Lounge",  :id 4, :exit {5});
  Room(:name "Kitchen", :id 3, :exit {5});
  Room(:name "Porch",   :id 5, :exit {6});
  Room(:name "Laundry", :id 6, :exit {});
END;
```

The CREATE statement instantiates a set of objects with the given values. In this example, each exit is represented only once. If there is an exit from a to b, we do not explicitly encode an exit from b to a, but handle the case using a rule.

```
RequirePath = RECORD
  from, to : INTEGER;
  moves : SEQUENCE OF INTEGER;
END;
```

A path goal is represented using a RequirePath object. A path between rooms is defined by the from and to attributes; the moves attribute is the sequence of rooms already visited.

A RequirePath goal is satisfied if from and to are identical, so the base case is handled by the following rule. The effect of this rule is to print the solution to the RequirePath goal.

```
(* A path from A to A is satisfied here by simply printing the path *)
RULE R1
IF
  RequirePath R { from <A> to <A> moves <M> }
THEN
  ShowPath(M)
END R1;
```

Given a set sequence of identifiers, ShowPath prints the names of the rooms in the order visited:

```
PROCEDURE ShowPath(moves : SEQUENCE OF INTEGER);
VAR m : INTEGER;
BEGIN
  FOREACH m IN moves DO
    PRINT FROM r : Room WHERE r.id = m SELECT r.name END;
  END;
  PRINTLN;
END ShowPath;
```

Where a RequirePath goal has different from and to attributes, the following rule is used:

```
(* To find a path from A to B, try finding a path from any exit of A *)
RULE R2
IF
  RequirePath { from <A> to <B> moves <M> : A # B } &
  Room { id <A> exit <<C>> : NOT (C IN M) }
THEN
  CREATE RequirePath(:from C, :to B, :moves M + [C]) END;
END R2;
```

The first pattern in the rule matches goals with different from and to attributes. The second pattern matches exits from the source room that have not already been visited. The effect of this rule is to create a sub-goal, which will search for a path from each connected room.

Since an exit from a to b implies an exit from b to a, the following rule is also required.

```
(* To find a path from A to B, try finding a path from any room that has
   exits to A *)
RULE R3
IF
  RequirePath { from <A> to <B> moves <M> : A # B } &
  Room { id <C> exit <<A>> : NOT (C IN M) }
THEN
  CREATE RequirePath(:from C, :to B, :moves M + [C]) END;
END R3;
```

Solutions to the problem are generated by asserting goals. For example:

```
RequirePath(:from 4, :to 6, :moves [4]);
System.Run();
```

```
> {Lounge}{Porch}{Laundry}
> {Lounge}{Hallway}{Kitchen}{Porch}{Laundry}
```

Note that this example does not explicitly remove sub-goals which may be required in some applications.

7.4 Hypotheses

Hypotheses are a way of managing alternative "possible worlds". A HYPOTHESIS is a set of objects that is held to be speculative, in the sense that they may or may not exist. Hypothetical objects must be explicitly tagged HYPOTHETICAL within their type declaration. Every hypothetical object exists within at least one hypothesis. A hypothetical object may only be referred to by other objects within the same hypothesis. Thus, any object that refers to a hypothetical object must be hypothetical. Hypothetical objects may refer to non-hypothetical objects.

A HYPOTHESIS has a special interpretation to the rule system. When a pattern (7.1) within a condition matches a hypothetical object, a "hypothetical context" is established. Further conditions only match hypothecial objects within the same hypothesis. In effect, every hypothetical pattern has an implied `hypothesis H` pattern binding, so all patterns must have the same value for their hypothesis attribute.

A new hypothesis is created using the HYPOTHESIS statement.

```
HypothesisStatement =
  "HYPOTHESIS" Ident [ "FROM" Expr ] StatementList "END".
```

A HYPOTHESIS statement creates a new hypothesis. By default, this is an empty hypothesis containing no hypothetical objects. If a hypothesis is specified using the FROM clause, the new hypothesis contains identical copies of all objects in the specified hypothesis. A statement sequence is executed in the new hypothesis. Within the statement, the MAP function can be used to refer to the objects in the new hypothesis that correspond to objects in the source hypothesis.

Example:

```
RULE R4
  IF
    ProximityOfTOI {entity <e> toi <t> } &
    Entity e { hypothesis <H> id unknown }
  THEN
    HYPOTHESIS h FROM H
      MAP(h, e).id := t.id;
    END;
END R4;
```

Within the HYPOTHESIS block, MAP(h,e).id denotes the id attribute of the object in hypothesis h which was derived from hypothetical object e within H.

7.5 Rule Scheduling

The current implementation of SDL uses the RETE algorithm to compute satisfiability for rules. Satisfiability is reevaluated whenever the attributes change in the patterns that comprise a rule. Once a rule has been fired, it does not fire again for a given set of objects *unless an attribute of one of these objects has changed.* At that point, satisfiability is reevaluated.

It is important to take care in the formulation of rules. Rules with conditions that are too general can lead to recursion. This results in the same rule firing repeatedly.

```
RULE Bad
  IF Complex c { x <X> : X > 10 }
  THEN
    c.y := X;
END Bad;
```

In the above example, the rule activates whenever a Complex object has x > 10. As a result, it changes y but this does not change the satisfiability of the rule, so the system is free to activate the rule again. A better formulation would be:

```
IF Complex c { x <X> y <Y> : X > 10 : Y # X }
```

8 Statements

8.1 Assignments

```
Becomes = ":=" | "BECOMES" | "BE".
AssignmentStatement =
  Designator Becomes [ Expr ].
```

Assignment statment replace the current value of a variable with a new value specified by an expression. The expression must be "assignment compatible" with the variable.

Examples:

```
i := 0;
class.name := "Oberon";
s := { red, green, blue };
```

The expression on the right hand side of the := symbol is not required where the designated object has unity type. This applies to fields of compound types declared without associated types. See (5.4.2).

8.2 Procedure Calls

```
ActualParam = [ ":" Ident ] Expr.
ActualParamList = ActualParam { "," ActualParam }.
```

A procedure call activates a procedure. It may contain a list of actual parameters which replace the corresponding formal parameters in the procedure declaration (9). The correspondence is established by the positions of the parameters in the actual and formal parameter lists. Each actual parameter must be an expression of the corresponding type, which is evaluated before the procedure is activated, and the resulting value is assigned to the formal parameter.

Optional parameters are specified by giving the parameter name before the expression, and are not required to be in any particular position or order. Optional parameters are not considered significant in the positioning of mandatory (non-optional) parameters.

```
d := Distance(p1, p2);

dNm := Distance(p1, p2, :units nauticalMiles);
dNm := Distaince(:units nauticalMiles, p1, p2);   (* equivalent *)
```

8.3 RETURN Statements

```
ReturnStatement =
  "RETURN" [ Expr ].
```

RETURN statements cause a procedure activation to terminate. If an expression is specified, the value of the expression is returned by the procedure. The type of the expression must match the return type of the procedure in which it is used.

8.4 IF Statements

```
IfStatement =
  "IF" Expr "THEN" StatementList
  { "ELSIF" Expr "THEN" StatementList }
  [ "ELSE" StatementList ] "END".
```

IF statements specify conditional execution of statement sequences. A Boolean expression called a "guard" precedes each statement. The guards are evaluated in sequence until one evaluates TRUE, whereupon its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the ELSE symbol is executed (if there is one).

See also: IF expressions (6.2).

8.5 WHILE Statements

```
WhileStatement =
  "WHILE" Expr "DO" StatementList "END"
```

WHILE statements specify the repeated executation of a statement while the Boolean expression (or "guard") evaluates true.

Example:

```
WHILE i > 0 DO
  i := i DIV 2; INC(power)
END
```

8.6 REPEAT Statements

```
RepeatStatement =
  "REPEAT" StatementList "UNTIL" Expr "END"
```

REPEAT statements specify the repeated executation of a statement until the Boolean expression evaluates true. The statement is always executed at least once.

8.7 FOR Statements

```
ForStatement =
 "FOR" VarIdent ":=" Expr "TO" Expr [ "BY" Expr ]
 "DO" StatementList
 "END".
```

FOR statements specify the repeated execution of a statement for each of a range of values assigned to an integer "control variable". The statement

```
FOR i := low TO high BY step DO statement END;
```

is equivalent to:

```
i := low; temp := high;
IF step > 0 THEN
  WHILE i <= temp DO statement; i := i + step END
ELSE
  WHILE i >= temp DO statement; i := i + step END
END;
```

Example:

```
sum := 0;
FOR i := 0 TO SIZE(s)-1 DO
  sum := sum + s[i]
END;
```

8.8 FOREACH Statements

```
ForEachStatement =
  "FOREACH" Ident "IN" Expr "DO" StatementList "END"
```

FOREACH statements specify the repeated execution of a statement for each of the values in a set or sequence, assigned to a variable of the set element type.

Example:

```
FOREACH v IN Vessel DO
  v.Move();
END;
```

8.9 CREATE Statements

```
CreateStatement =
  "CREATE"
  { [ "[" Ident "]" ] Initialisation [ ";" ] }
  "END"
```

CREATE statements create new instances of compound types (5.4). Each statement specifies the name of a type, and optionally the name of a symbol to which the new object is bound. The binding is valid only within the CREATE statement, and is intended to simplify the creation of graphs containing acyclic references. An initialisation clause specifies the values of fields of the new object.

See also: NEW (6.3), DELETE (8.12).

8.10 WITH Statements

```
WithPart = Ident ":" Qualident "DO" StatementList .

WithStatement =
  "WITH"
   WithPart { "|" WithPart }
   [ "ELSE" StatementList ]
  "END" .
```

The with statement executes a statement sequence depending on the result of a type test. If v refers to a record with static type T0, the statement

```
WITH v : T1 DO S1 | v : T2 DO S2 ELSE S3 END
```

has the following meaning. If v has dynamic type T1 then S1 is executed with v being regarded as having static type T1. Otherwise, if v has dynamic type T2 then S2 is execcuted with v regarded as having static type T2. Otherwise S3 is executed.

8.11 TEMPORAL Statements

The TEMPORAL statement asserts a set of temporal constraints (5.5.1). The constraints have the same format as temporal queries (expressions).

8.12 DELETE Statements

```
DeleteStatement =
  "DELETE" Expr.
```

DELETE statements are used to remove objects from the data base. The effect of the delete operation is to remove an object from the instance set of all its base types. This means that it does not match patterns in rules, and does not appear as an instance of any base type. However, the object *remains in the data base* until all references to the object are removed, whereupon it may be reclaimed by the system garbage collector.

See also: NEW (6.3), CREATE (8.9).

8.13 PRINTLN Statements

```
PrintStatement =
  ("PRINT" | "PRINTLN") ElementList.
```

Two statements are provided for simple text output. The statements specify a list of expressions which are evaluated and printed as text to standard output. The PRINTLN statement terminates its output with a new-line. The PRINT statement does not terminate output, so multiple PRINT statements can be used to compose a line of text.

Example: see USER (11).

8.14 UPDATE Statements

```
UpdateStatement =
  "UPDATE" StatementList "END".
```

Whenever objects are modified the run-time system updates internal information about which rules are satisfied by the current data base state. This is done in a two-stage process. Rule activations related to the old state of an object are retracted, then new activations are computed for the new state of the object. When many changes are made to a set of objects this can result in wasted computation.

Within an UPDATE statement, retractions are processed immediately but assertions are deferred until the end of the UPDATE statement. If UPDATEs are nested, no assertions occur until the *outermost* UPDATE exits.

9 Procedure Declarations

```
FormalParam = [ "OPTIONAL" ] IdentList ":" Type .

FormalParamsProper = "(" [ FormalParam { ";" FormalParam } ] ")".

FormalParams = FormalParamsProper [ ":" Type ] .

Receiver = "(" Ident ":" Ident ")".

ProcDecl =
  [ "ABSTRACT" ] "PROCEDURE"
  [ Receiver ] Ident
  [ FormalParams ] ";"
  DeclList
  [ "BEGIN" StatementList ]
  "END" Ident.
```

A "procedure declaration" consists of a procedure heading and a procedure body. The procedure heading defines an identifier for the procedure, and its "formal parameters". The procedure body includes a list of local declarations, and an optional sequence of statements. The procedure identifier is repeated at the end of the declaration.

There are two kinds of procedures: "proper procedures" and "function procedures". Function procedures are activated by a function designator as part of an expression. The result of the function procedure is defined by its return statement. Proper procedures are activated by a procedure call within a statement.

Example:

```
PROCEDURE Min (s : SEQUENCE OF INTEGER) : INTEGER;
(* find the smallest element in a sequence of integers *)
VAR i, min : INTEGER;
BEGIN
  min := s[0];
  FOR i := 1 TO LEN(s)-1 DO
    IF s[i] < min THEN min := s[i] END
  END;
  RETURN min;
END Min;
```

9.1 Formal Parameters

Formal parameters are identifiers declared in the formal parameter list of a procedure. They correspond to the actual parameters specified in the call to the procedure. The formal parameter list is enclosed within parentheses. A procedure with no parameters must have an empty parameter list. The return type for function procedures is specified following the parameter list.

Parameters may be specified as OPTIONAL, which means that they may or may not be given in a call to the procedure. The PARAM function can be used to test if a particular parameter is given. In the actual parameter list, every optional parameter must be preceded by a colon followed by its identifier.

Example:

```
PROCEDURE Inc(i : INTEGER; OPTIONAL by : INTEGER) : INTEGER;
(* return the result of incrementing <i>.
   Optionally, an amount may be given *)
BEGIN
  IF PARAM(by) THEN
    RETURN i + by;
  ELSE
    RETURN i + 1;
  END;
END Inc;
...
result := Inc(10);
result := Inc(10, :by 2);   (* optional parameter given *)
```

9.2 Type-bound Procedures

Procedures may be associated with compound data types. The procedures are said to be "bound" to the data type. The binding is expressed by the type of the "receiver" in the heading of a procedure declaration. The receiver may be any compound type T.

If procedure P is bound to type T0, it is also implicitly bound to any type T1 which is an extension of T0. However, a procedure Q (with the same name as P) may be explicitly bound to T1 in which case it overrides the binding of P. Q is considered to be a "redefinition" of P for T1.

A procedure bound to type T may be declared ABSTRACT, which means that it *must* be overridden in any extension of T. Abstract procedures may only be bound to abstract types.

10 Module Declaration

```
DeclList = (
  [ "HINT" { HintDecl ";" } ]
  { "TYPE" { TypeDecl ";" } | "CONST" { ConstDecl ";" } }
  [ "VAR" { VarDecl ";" } ]
  { ProcDecl ";" }
).

Import = Ident [ ":=" Ident ] .

ModuleDecl =
```

```
"MODULE" Ident ";"
[ "IMPORT" Import { "," Import } ";" ]
DeclList
{ RuleDecl ";" }
[ "BEGIN" StatementList ]
"END" Ident "."
```

A module is a collection of declarations of constants, types, procedures, variables and rules. A module includes a sequence of statements used to initialise the module to a valid state.

Each module defines its own set of identifiers. A module may import declarations from other modules. Identifiers `id` in an imported module `M` is referred to in importing modules as `M.id`. An imported module may be renamed using the `:=` clause in the import declaration. This associates the given name with the module, instead of the name of the module itself.

11 USER Declaration

```
User = "USER" { Statement ";" } "END".
```

`USER` declarations identify statements that are issued interactively by the user. Each statement in a `USER` declaration is executed immediately before the next statement is compiled.

Example:

```
  java SDL.Comp stdin
  USER
  PRINTLN 1 + 2 * 3 + 4;
> 11
  PRINTLN "Hello " + "There";
> Hello There
  PRINTLN SIZE("Hello " + "There");
> 11
  END.
```

See also: `PRINTLN` (8.13).

12 Conclusion

SDL is a Situation Description Language intended for use in situation assessment problems. SDL provides knowledge modelling and inference facilities for reasoning with information.

SDL includes an object-oriented data model with single inheritance. It provides a forward-chaining inference system using a RETE-based pattern matcher. SDL includes special data types to deal with time, space, and uncertainty. A hypothesis system allows the system to deal with multiple concurrent hypotheses in a systematic way.

Many of these facilities are implementable within other language systems. However, SDL provides consistent support at the *language* level which means that code written within SDL is more clear and expressive when dealing with these complex abstract concepts. This helps to reduce the semantic mismatch between knowledge and its representation within a computer system.

References

1. Greenhill, S., Venkatesh, S., Pearce, A. & Ly, T. (2002) *SDL Implementation Report*, Technical Report Submitted for publication, DSTO, Melbourne, Australia.

2. Ly, T., Greenhill, S., Venkatesh, S. & Pearce, A. (2002) *Submariner Situation Assessment*, Technical Report Submitted for publication, DSTO, Melbourne, Australia.

Appendix A EBNF Grammar for SDL

This section gives a complete definition of the syntax of SDL using EBNF notation (see 1).

```
CHARACTERS

  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".

  digit = "0123456789".

  tab = CHR(9).

  cr = CHR(13).

  lf = CHR(10).

  eol  = cr.

  singleQuote = "'".

  doubleQuote = '"'.

  bodySingleString = ANY - singleQuote.

  bodyDoubleString = ANY - doubleQuote.

TOKENS

  ident = letter { digit | letter }.

  varIdent = "?" letter {digit | letter }.

  number = digit { digit}.

  realNumber =
    digit {digit} "." {digit} [ "E" [ "+" | "-"] digit {digit} ].

  string = singleQuote { bodySingleString } singleQuote
  | doubleQuote { bodyDoubleString } doubleQuote.

IGNORE eol + lf + tab

COMMENTS FROM "(*" TO "*)" NESTED

PRODUCTIONS

Ident = ident .

Qualident = [Ident  "." ] Ident .

JavaClassIdent = Ident { "." Ident }.
```

```
IdentList = Ident { "," Ident } .

Fields = [ IdentList [ ":" Type] [ "DEFAULT" Expr ] ] .

FieldList = Fields { ";" Fields }.

CompoundType
=
   { "ABSTRACT"
   | "HYPOTHETICAL"
   | "INSTANT"
   | "INTERVAL"
   | "PERSISTENT"
   }

   ( "RECORD" | "VARIANT")

   [ "(" Qualident ")" ]

   FieldList
  "END" .

Type
=
    Qualident
  | CompoundType
  | "(" IdentList ")"
  | "SET" "OF" Type
  | "SEQUENCE" "OF" Type
  | "POTENTIAL" Type
  | "STRING"
  | "CHAR"
  | "INTEGER"
  | "REAL"
  | "BOOLEAN"
  | "JAVA" JavaClassIdent
  | "SYMBOL"
  | "SPATIAL"
  | "POINT"
  | "LINE"
  | "VECTOR" | "REGION" | "EXTENT" | "CIRCLE"
  | "LENGTH"
.

FieldSelector =  "." Ident .
```

```
ArraySelector =  "[" Expr "]" .

FunctionSelector = "(" [ ActualParam { "," ActualParam } ] ")" .

TypeSelector = "{" Type "}" .

Selector
= FieldSelector
  | ArraySelector
  | FunctionSelector
  | TypeSelector .

InnerDesignator
=
  ( Qualident
  | "NEW" Initialisation
  | "MAP" "(" Expr "," Expr ")"
  | "REL" "(" AbsoluteTime "," AbsoluteTime [ "," Expr ] ")"
  )
  { Selector } .

Designator = InnerDesignator .

PotentialElement = Expr "CF" Expr .

PotentialElementList = [ PotentialElement { "," PotentialElement } ] .

PotentialSet = "{{" PotentialElementList "}}" .

Element = Expr .

ElementList = [ Element { "," Element } ] .

Set = "{" ElementList "}" .

Sequence = "[" ElementList "]" .

UnaryExpr =
  Designator
  | "(" Expr ")"
  | number
  | realNumber
  | string
  | Set
  | Sequence
  | PotentialSet
  | "TRUE"
```

```
  | "FALSE"
  | "NIL"
  | ExistExpr
  | TimeDate
  | "-" UnaryExpr
  | "NOT" UnaryExpr
  | "LIKELY" UnaryExpr
  | "UNLIKELY" UnaryExpr
  | "SIZE" "(" Expr ")"
  | "CHR" "(" Expr ")".

Initialisation
=
  ( "JAVA" JavaClassIdent | Qualident ) "("
    InitialParam
    { "," InitialParam }
  ")" .

Factor =
    UnaryExpr
  | "IF" Expr
    "THEN" Expr
    "ELSE" Expr "END"
  | "PARAM" "(" Ident ")"
  | SelectExpr
  | TemporalStat .

MulOp =
    "*"
  | "TIMES"
  | "MULTIPLIED" "BY"
  | "/"
  | "DIVIDED" "BY"
  | "DIV"
  | "MOD" .

MulExpr = Factor { MulOp Factor }.

AddOp
=
    "+"
  | "PLUS"
  | "-"
  | "MINUS" .

AddExpr = MulExpr { AddOp MulExpr }.
```

```
RelOp =
    "<"
  | "LESS" "THAN"
  | "<="
  | ">"
  | "GREATER" "THAN"
  | ">=" .

RelExpr = AddExpr [
    RelOp AddExpr
  | "IN" AddExpr
  | "IS" Type
  | "HAS" Ident ].

EquOp =
    "="
  | "EQUAL" "TO"
  | "EQUALS"
  | "#"
  | "NOT" "EQUAL" "TO" .

EquExpr = RelExpr [ EquOp RelExpr ].

AndExpr = EquExpr { "AND" EquExpr }.

BoolExpr = AndExpr { "OR" AndExpr }.

VerboseExpr
=
  BoolExpr
  | "POINT" VerboseExpr
    "," VerboseExpr
  | "LINE" "FROM" VerboseExpr
    "TO" VerboseExpr
    { "TO" VerboseExpr
    } "END"
  | "CLOSEST" "TO" VerboseExpr "OF" VerboseExpr
  | "LENGTH" "OF" VerboseExpr
  | "DISTANCE" "FROM" VerboseExpr "TO" VerboseExpr
  | "BEARING" "FROM" VerboseExpr "TO" VerboseExpr
.

Expr = VerboseExpr { ("INSIDE" | "OUTSIDE") VerboseExpr}.

SelectWhere = [ "LET" Ident Becomes ] Expr .

SelectFrom = Ident ":" Expr .
```

```
FromWhere
=
  SelectFrom { "," SelectFrom }
  [ "WHERE" SelectWhere { "," SelectWhere } ]
.

SelectExpr
= "FROM" FromWhere
  "SELECT" Expr
  "END"
.

ExistExpr
= ( "EXISTS" | "ALL" )
  FromWhere
  "END"
.

ConstExpr = Expr.

TemporalDesignator
= UnaryExpr
.

TemporalAnchor
= (
    "STARTS"
  | "ENDS"
  | "HAPPENS"
).

Time
= Expr
  {":" Expr
  } TemporalUnit
.

TimeDate = ("@" | "~") Time.

AbsoluteTime
=
  ( TemporalDesignator [TemporalAnchor
    ]
  | "START" "OF" TemporalDesignator
  | "END" "OF" TemporalDesignator
  ).
```

```
TemporalUnit
= (
    ("MILLISECOND" | "MS")
  | ("SECOND" | "SECONDS" )
  | ("MINUTE" | "MINUTES" )
  | ("HOUR" | "HOURS" )
  | ("DAY" | "DAYS" )
  ).

TemporalRange
= ( ["EXACTLY"] AddExpr TemporalUnit
  | "RANGE" Expr [ TemporalUnit ] "TO" Expr TemporalUnit
  | "MORE" "THAN" AddExpr TemporalUnit
  | "LESS" "THAN" AddExpr TemporalUnit
  ).

TemporalDistance
= ( TemporalRange ).

TemporalConstraint
= ( TemporalAnchor
    (
      ("AT" | "AS") AbsoluteTime
    | [TemporalDistance]
      ( "AFTER"
      | "BEFORE"
      )
      AbsoluteTime
    | "DURING" TemporalDesignator
    | "BETWEEN" AbsoluteTime "AND" AbsoluteTime
    | "WITHIN" Expr TemporalUnit "OF" AbsoluteTime
    )
  | "HAS" "DURATION" TemporalRange
  ).

PatternBinding = Ident ( "<<" Ident ">>" | "<" Ident ">" | Expr).

Pattern = Qualident [ Ident ] "{" { PatternBinding | ":" Expr } "}" .

Term = "(" Cond ")" | Pattern .

NotCond = [ "~" ] Term .

AndCond = NotCond { "&" NotCond }.

OrCond = AndCond { "|" AndCond }.
```

```
Cond = OrCond.

CondList = Cond.

ForwardRule = "IF" CondList "THEN" RuleStatementList .

EventRule =
  "EVENT" Pattern "WHEN" Cond
  [ "ACTIVE" TemporalConstraints ]
  [ "INACTIVE" TemporalConstraints ] .

RuleDecl = "RULE" Ident (ForwardRule | EventRule) "END" Ident .

TypeDecl = Ident "=" Type .

ConstDecl = Ident "=" ConstExpr .

VarDecl = IdentList ":" Type .

HintDecl = JavaClassIdent "(" Ident ")" ":" Type .

DeclList
= (
    [ "HINT" { HintDecl ";" } ]
    { "TYPE" { TypeDecl ";" } | "CONST" { ConstDecl ";" } }
    [ "VAR" { VarDecl ";" } ]
    { ProcDecl ";" }
  ).

ActualParam = [ ":" Ident ] Expr .

InitialParam = [":" Ident] [ Expr ] .

FormalParam = [ "OPTIONAL" ] IdentList ":" Type .

FormalParamsProper = "(" [ FormalParam { ";" FormalParam } ] ")".

FormalParams = FormalParamsProper [ ":" Type ] .

Receiver = "(" Ident ":" Ident ")".

TemporalConstraints = TemporalConstraint { "ALSO" TemporalConstraint } .

TemporalStat =
  "TEMPORAL"
  [ "IN" "(" Expr
```

```
  ")" ] {
  TemporalDesignator
  TemporalConstraints
  ";"
  } "END"
.

Becomes = ":=" | "BECOMES" | "BE".

WithPart = Ident ":" Qualident "DO" StatementList .

WithStatement =
  "WITH"
   WithPart { "|" WithPart }
   [ "ELSE" StatementList ]
  "END" .

Statement
=
  [ Designator [ Becomes [ Expr ] ]
  | ("PRINT" | "PRINTLN") ElementList
  | "IF" Expr "THEN" StatementList
    { "ELSIF" Expr "THEN" StatementList
    }
    [ "ELSE" StatementList
    ] "END"
  | "UPDATE" StatementList "END"
  | "FOR" Ident Becomes Expr
    "TO" Expr [ "BY" Expr ]
    "DO" StatementList "END"
  | "FOREACH" Ident
    "IN" Expr
    "DO" StatementList "END"
  | "WHILE" Expr
    "DO" StatementList "END"
  | "RETURN" [ Expr ]
  | "DELETE" Expr
  | WithStatement
  | "CREATE"
    { [ "[" Ident "]" ] Initialisation
      [ ";" ]
    }
    "END"
  | "HYPOTHESIS" Ident [ "FROM" Expr ]
      StatementList
    "END"
  | TemporalStat
```

```
  ].

RuleStatement = Statement.
StatementList = Statement { ";" Statement } .

RuleStatementList = RuleStatement { ";" RuleStatement } .

ProcDecl
=
  [ "ABSTRACT" ] "PROCEDURE"
  [ Receiver ] Ident
  [ FormalParams ] ";"
  DeclList
  [ "BEGIN" StatementList ]
  "END" Ident
.

Import = Ident [ ":=" Ident ] .

ModuleDecl
=
  "MODULE" Ident ";"
  [ "IMPORT" Import { "," Import } ";" ]
  DeclList
  { RuleDecl ";" }
  [ "BEGIN" StatementList ]
  "END" Ident "."
.

User = "USER" { Statement ";" } "END".

SDL = [ ModuleDecl ] [ User ].

END SDL.
```

DISTRIBUTION LIST

Situation Description Language

S. Greenhill and S.Venkatesh and A. Pearce and T.C. Ly

	Number of Copies
DEFENCE ORGANISATION	
S&T Program	
Chief Defence Scientist	1
FAS Science Policy	
AS Science Corporate Management	
Director General Science Policy Development	
Counsellor, Defence Science, London	Doc Data Sht
Counsellor, Defence Science, Washington	Doc Data Sht
Scientific Adviser to MRDC, Thailand	Doc Data Sht
Scientific Adviser Joint	1
Navy Scientific Adviser	Doc Data Sht
Scientific Adviser, Army	Doc Data Sht
Air Force Scientific Adviser	1
Director Trials	1
Information Sciences Laboratory	
Don Perugini	1
Poh Lian Choong	1
Systems Sciences Laboratory	
Chief of Maritime Operation Division	1
Research Leader Combat Information Systems	1
Head Submarine Combat System	1
John Best	1
Thanh Chi Ly	1
Chris Davis	1
Simon Goss	1
DSTO Library and Archives	
Library, Stirling	1
Australian Archives	1
Capability Systems Staff	
Director General Maritime Development	Doc Data Sht
Knowledge Staff	
Director General Command, Control, Communications and Computers (DGC4)	Doc Data Sht

Army

ABCA National Standardisation Officer, Land Warfare Development Sector, Puckapunyal	4

Intelligence Program

DGSTA, Defence Intelligence Organisation	1
Manager, Information Centre, Defence Intelligence Organisation	1

Defence Libraries

Library Manager, DLS-Canberra	1
Library Manager, DLS-Sydney West	Doc Data Sht

UNIVERSITIES AND COLLEGES

Australian Defence Force Academy Library	1
Hargrave Library, Monash University	Doc Data Sht
Librarian, Flinders University	1
Curtin Library, Curtin University	1

School of Computing, Curtin University

Stewart Greenhill	1
Svetha Venkatesh	1

University of Melbourne

Adrian Pearce	1

OTHER ORGANISATIONS

National Library of Australia	1
NASA (Canberra)	1
AusInfo	1

INTERNATIONAL DEFENCE INFORMATION CENTRES

US Defense Technical Information Center	2
UK Defence Research Information Centre	2
Canada Defence Scientific Information Service	1
NZ Defence Information Centre	1

ABSTRACTING AND INFORMATION ORGANISATIONS

Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
Materials Information, Cambridge Scientific Abstracts, US	1
Documents Librarian, The Center for Research Libraries, US	1

INFORMATION EXCHANGE AGREEMENT PARTNERS

Acquisitions Unit, Science Reference and Information Service, UK	1
Library – Exchange Desk, National Institute of Standards and Technology, US	1

SPARES

DSTO Edinburgh Library	5

Total number of copies:	48

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA				1. CAVEAT/PRIVACY MARKING
2. TITLE Situation Description Language			3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)	
4. AUTHORS S. Greenhill and S.Venkatesh and A. Pearce and T.C. Ly			5. CORPORATE AUTHOR Systems Sciences Laboratory PO Box 1500 Edinburgh, South Australia, Australia 5111	
6a. DSTO NUMBER DSTO–GD–0332	6b. AR NUMBER 012-415		6c. TYPE OF REPORT General Document	7. DOCUMENT DATE September, 2002
8. FILE NUMBER M9505/23/31	9. TASK NUMBER LRR 98/081	10. SPONSOR	11. No OF PAGES 50	12. No OF REFS 2
13. URL OF ELECTRONIC VERSION http://www.dsto.defence.gov.au/corporate/reports/DSTO–GD–0332.pdf			14. RELEASE AUTHORITY Chief, Maritime Operations Division	

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved For Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DEFTEST DESCRIPTORS

19. ABSTRACT

SDL is a Situation Description Language intended for use in situation assessment problems. SDL provides knowledge modelling and inference facilities for reasoning with information.

The SDL includes an object-oriented data model with single inheritance. It provides a forward-chaining inference system using a RETE-based pattern matcher. SDL includes special data types to deal with time, space, and uncertainty. A hypothesis system allows the system to deal with multiple concurrent hypotheses in a systematic way.

This document provides details required by programmers or knowledge engineers intending to use the SDL system.

DEFENCE
SCIENCE & TECHNOLOGY

SYSTEMS SCIENCES LABORATORY
PO BOX 1500 EDINBURGH SOUTH AUSTRALIA 5111 AUSTRALIA
TELEPHONE (08) 8259 5555